

# Quarkus - Amazon SNS Client

Amazon Simple Notification Service (SNS) is a highly available and fully managed pub/sub messaging service. It provides topics for high-throughput, push-based, many-to-many messaging. Messages can fan out to a large number of subscriber endpoints for parallel processing, including Amazon SQS queues, AWS Lambda functions, and HTTP/S webhooks. Additionally, SNS can be used to fan out notifications to end users using mobile push, SMS and email.

You can find more information about SNS at [the Amazon SNS website](#).



The SNS extension is based on [AWS Java SDK 2.x](#). It's a major rewrite of the 1.x code base that offers two programming models (Blocking & Async).

This technology is considered preview.



In *preview*, backward compatibility and presence in the ecosystem is not guaranteed. Specific improvements might require to change configuration or APIs and plans to become *stable* are under way. Feedback is welcome on our [mailing list](#) or as issues in our [GitHub issue tracker](#).

For a full list of possible extension statuses, check our [FAQ entry](#).

The Quarkus extension supports two programming models:

- Blocking access using URL Connection HTTP client (by default) or the Apache HTTP Client
- [Asynchronous programming](#) based on JDK's `CompletableFuture` objects and the Netty HTTP client.

In this guide, we see how you can get your REST services to use SNS locally and on AWS.

## Prerequisites

To complete this guide, you need:

- JDK 1.8+ installed with `JAVA_HOME` configured appropriately
- an IDE
- Apache Maven 3.6.3
- An AWS Account to access the SNS service
- Optionally, Docker for your system to run SNS locally for testing purposes

## Set up SNS locally

The easiest way to start working with SNS is to run a local instance as a container.

```
docker run -it --publish 8009:4575 -e SERVICES=sns -e START_WEB=0
localstack/localstack:0.11.1
```

This starts a SNS instance that is accessible on port **8009**.

Create an AWS profile for your local instance using AWS CLI:

```
$ aws configure --profile localstack
AWS Access Key ID [None]: test-key
AWS Secret Access Key [None]: test-secret
Default region name [None]: us-east-1
Default output format [None]:
```

## Create a SNS topic

Create a SNS topic using AWS CLI and store in **TOPIC\_ARN** environment variable

```
TOPIC_ARN=`aws sns create-topic --name=QuarksCollider --profile
localstack --endpoint-url=http://localhost:8009`
```

If you want to run the demo using SNS on your AWS account, you can create a topic using AWS default profile

```
TOPIC_ARN=`aws sns create-topic --name=QuarksCollider`
```

## Solution

The application built here allows to shoot elementary particles (quarks) into a **QuarksCollider** topic of the AWS SNS. Additionally, we create a resource that allows to subscribe to the **QuarksCollider** topic in order to receive published quarks.

We recommend that you follow the instructions in the next sections and create the application step by step. However, you can go right to the completed example.

Clone the Git repository: `git clone https://github.com/quarkusio/quarkus-quickstarts.git`, or download an [archive](#).

The solution is located in the **amazon-sns-quickstart** directory.

# Creating the Maven project

First, we need a new project. Create a new project with the following command:

```
mvn io.quarkus:quarkus-maven-plugin:1.5.1.Final:create \
    -DprojectId=org.acme \
    -DprojectArtifactId=amazon-sns-quickstart \
    -DclassName="org.acme.sns.QuarksCannonSyncResource" \
    -Dpath="/sync-cannon" \
    -Dextensions="resteasy-jsonb,amazon-sns,resteasy-mutiny"
cd amazon-sns-quickstart
```

This command generates a Maven structure importing the RESTEasy/JAX-RS, Mutiny and Amazon SNS Client extensions. After this, the `amazon-sns` extension has been added to your `pom.xml` as well as the Mutiny support for RESTEasy.

## Creating JSON REST service

In this example, we will create an application that allows to publish quarks. The example application will demonstrate the two programming models supported by the extension.

First, let's create the `Quark` bean as follows:

```

package org.acme.sns.model;

import io.quarkus.runtime.annotations.RegisterForReflection;
import java.util.Objects;

@RegisterForReflection
public class Quark {

    private String flavor;
    private String spin;

    public Quark() {
    }

    public String getFlavor() {
        return flavor;
    }

    public void setFlavor(String flavor) {
        this.flavor = flavor;
    }

    public String getSpin() {
        return spin;
    }

    public void setSpin(String spin) {
        this.spin = spin;
    }

    @Override
    public boolean equals(Object obj) {
        if (!(obj instanceof Quark)) {
            return false;
        }

        Quark other = (Quark) obj;

        return Objects.equals(other.flavor, this.flavor);
    }

    @Override
    public int hashCode() {
        return Objects.hash(this.flavor);
    }
}

```

Then, create a `org.acme.sns.QuarksCannonSyncResource` that will provide an API to shoot quarks into the SNS topic via the SNS synchronous client.

```

package org.acme.sns;

import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.databind.ObjectWriter;
import javax.inject.Inject;
import javax.ws.rs.Consumes;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
import org.acme.sns.model.Quark;
import org.eclipse.microprofile.config.inject.ConfigProperty;
import org.jboss.logging.Logger;
import software.amazon.awssdk.services.sns.SnsClient;
import software.amazon.awssdk.services.sns.model.PublishResponse;

@Path("/sync/cannon")
@Produces(MediaType.TEXT_PLAIN)
public class QuarksCannonSyncResource {

    private static final Logger LOGGER = Logger.getLogger(
        QuarksCannonSyncResource.class);

    @Inject
    SnsClient sns;

    @ConfigProperty(name = "topic.arn")
    String topicArn;

    static ObjectWriter QUARK_WRITER = new ObjectMapper().
        writerFor(Quark.class);

    @POST
    @Path("/shoot")
    @Consumes(MediaType.APPLICATION_JSON)
    public Response publish(Quark quark) throws Exception {
        String message = QUARK_WRITER.writeValueAsString(quark);
        PublishResponse response = sns.publish(p -> p.topicArn
            (topicArn).message(message));
        LOGGER.infov("Fired Quark[{0}, {1}]", quark.getFlavor(),
            quark.getSpin());
        return Response.ok().entity(response.messageId()).build();
    }
}

```

Because of the fact that messages published must be simply a `String` we're using Jackson's `ObjectWriter` in order to serialize our `Quark` objects into a `String`.

The missing piece is the subscriber that will receive the messages published to our topic. Before implementing subscribers, we need to define POJO classes representing messages posted by the AWS SNS.

Let's create two classes that represent SNS Notification and SNS Subscription Confirmation messages based on the [AWS SNS Message and JSON formats](#)

Create `org.acme.sns.model.SnsNotification` class

```
package org.acme.sns.model;

import com.fasterxml.jackson.annotation.JsonIgnoreProperties;
import com.fasterxml.jackson.annotation.JsonProperty;

@JsonIgnoreProperties(ignoreUnknown = true)
public class SnsNotification {

    @JsonProperty("Message")
    private String message;

    @JsonProperty("MessageId")
    private String messageId;

    @JsonProperty("Signature")
    private String signature;

    @JsonProperty("SignatureVersion")
    private String signatureVersion;

    @JsonProperty("SigningCertURL")
    private String signingCertUrl;

    @JsonProperty("Subject")
    private String subject;

    @JsonProperty("Timestamp")
    private String timestamp;

    @JsonProperty("TopicArn")
    private String topicArn;

    @JsonProperty("Type")
    private String type;

    @JsonProperty("UnsubscribeURL")
    private String unsubscribeURL;

    public String getMessage() {
        return message;
    }
}
```

```
public void setMessage(String message) {
    this.message = message;
}

public String getMessageId() {
    return messageId;
}

public void setMessageId(String messageId) {
    this.messageId = messageId;
}

public String getSignature() {
    return signature;
}

public void setSignature(String signature) {
    this.signature = signature;
}

public String getSignatureVersion() {
    return signatureVersion;
}

public void setSignatureVersion(String signatureVersion) {
    this.signatureVersion = signatureVersion;
}

public String getSigninCertUrl() {
    return signinCertUrl;
}

public void setSigninCertUrl(String signinCertUrl) {
    this.signinCertUrl = signinCertUrl;
}

public String getSubject() {
    return subject;
}

public void setSubject(String subject) {
    this.subject = subject;
}

public String getTimestamp() {
    return timestamp;
}

public void setTimestamp(String timestamp) {
```

```

        this.timestamp = timestamp;
    }

    public String getTopicArn() {
        return topicArn;
    }

    public void setTopicArn(String topicArn) {
        this.topicArn = topicArn;
    }

    public String getType() {
        return type;
    }

    public void setType(String type) {
        this.type = type;
    }

    public String getUnsubscribeURL() {
        return unsubscribeURL;
    }

    public void setUnsubscribeURL(String unsubscribeURL) {
        this.unsubscribeURL = unsubscribeURL;
    }
}

```

Then, create `org.acme.sns.SnsSubscriptionConfirmation`

```

package org.acme.sns.model;

import com.fasterxml.jackson.annotation.JsonProperty;

public class SnsSubscriptionConfirmation {

    @JsonProperty("Message")
    private String message;

    @JsonProperty("MessageId")
    private String messageId;

    @JsonProperty("Signature")
    private String signature;

    @JsonProperty("SignatureVersion")
    private String signatureVersion;

    @JsonProperty("SigningCertURL")

```



```

private String signingCertUrl;

@JsonProperty("SubscribeURL")
private String subscribeUrl;

@JsonProperty("Timestamp")
private String timestamp;

@JsonProperty("Token")
private String token;

@JsonProperty("TopicArn")
private String topicArn;

@JsonProperty("Type")
private String type;

public String getMessage() {
    return message;
}

public void setMessage(String message) {
    this.message = message;
}

public String getMessageId() {
    return messageId;
}

public void setMessageId(String messageId) {
    this.messageId = messageId;
}

public String getSignature() {
    return signature;
}

public void setSignature(String signature) {
    this.signature = signature;
}

public String getSignatureVersion() {
    return signatureVersion;
}

public void setSignatureVersion(String signatureVersion) {
    this.signatureVersion = signatureVersion;
}

public String getSigningCertUrl() {

```

```

        return signingCertUrl;
    }

    public void setSigningCertUrl(String signingCertUrl) {
        this.signingCertUrl = signingCertUrl;
    }

    public String getSubscribeUrl() {
        return subscribeUrl;
    }

    public void setSubscribeUrl(String subscribeUrl) {
        this.subscribeUrl = subscribeUrl;
    }

    public String getTimestamp() {
        return timestamp;
    }

    public void setTimestamp(String timestamp) {
        this.timestamp = timestamp;
    }

    public String getToken() {
        return token;
    }

    public void setToken(String token) {
        this.token = token;
    }

    public String getTopicArn() {
        return topicArn;
    }

    public void setTopicArn(String topicArn) {
        this.topicArn = topicArn;
    }

    public String getType() {
        return type;
    }

    public void setType(String type) {
        this.type = type;
    }
}

```

Now, create `org.acme.QuarksShieldSyncResource` REST resources that: - Allows to subscribe

itself to our SNS topic - Unsubscribe from the SNS topic - Receive notifications from the subscribed SNS topic



Keep in mind that AWS SNS supports multiple types of subscribers (that is web servers, email addresses, AWS SQS queues, AWS Lambda functions, and many more), but for the sake of the quickstart we will show how to subscribe an HTTP endpoint served by our application.

```
package org.acme.sns;

import com.fasterxml.jackson.core.JsonProcessingException;
import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.databind.ObjectReader;
import java.util.HashMap;
import java.util.Map;
import javax.inject.Inject;
import javax.ws.rs.Consumes;
import javax.ws.rs.HeaderParam;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
import org.acme.sns.model.Quark;
import org.acme.sns.model.SnsNotification;
import org.acme.sns.model.SnsSubscriptionConfirmation;
import org.eclipse.microprofile.config.inject.ConfigProperty;
import org.jboss.logging.Logger;
import software.amazon.awssdk.services.sns.SnsClient;
import software.amazon.awssdk.services.sns.model.SubscribeResponse;

@Path("/sync/shield")
public class QuarksShieldSyncResource {

    private static final Logger LOGGER = Logger.getLogger(
        QuarksShieldSyncResource.class);

    private static final String NOTIFICATION_TYPE = "Notification";
    private static final String SUBSCRIPTION_CONFIRMATION_TYPE =
        "SubscriptionConfirmation";
    private static final String UNSUBSCRIPTION_CONFIRMATION_TYPE =
        "UnsubscribeConfirmation";

    @Inject
    SnsClient sns;

    @ConfigProperty(name = "topic.arn")
    String topicArn;

    @ConfigProperty(name = "quarks.shield.base.url")
```

```

String quarksShieldBaseUrl;

private volatile String subscriptionArn;

static Map<Class<?>, ObjectReader> READERS = new HashMap<>();

static {
    READERS.put(SnsNotification.class, new ObjectMapper()
.readerFor(SnsNotification.class));
    READERS.put(SnsSubscriptionConfirmation.class, new
ObjectMapper().readerFor(SnsSubscriptionConfirmation.class));
    READERS.put(Quark.class, new ObjectMapper().readerFor(
Quark.class));
}

@POST
@Consumes({MediaType.TEXT_PLAIN})
public Response notificationEndpoint(@HeaderParam("x-amz-sns-
message-type") String messageType, String message) throws
JsonProcessingException {
    if (messageType == null) {
        return Response.status(400).build();
    }

    if (messageType.equals(NOTIFICATION_TYPE)) {
        SnsNotification notification = readObject
(SnsNotification.class, message);
        Quark quark = readObject(Quark.class, notification
.getMessage());
        LOGGER.infov("Quark[{0}, {1}] collision with the
shield.", quark.getFlavor(), quark.getSpin());
    } else if (messageType.equals
(SUBSCRIPTION_CONFIRMATION_TYPE)) {
        SnsSubscriptionConfirmation subConf = readObject
(SnsSubscriptionConfirmation.class, message);
        sns.confirmSubscription(cs -> cs.topicArn(topicArn)
.token(subConf.getToken()));
        LOGGER.info("Subscription confirmed. Ready for quarks
collisions.");
    } else if (messageType.equals
(UNSUBSCRIPTION_CONFIRMATION_TYPE)) {
        LOGGER.info("We are unsubscribed");
    } else {
        return Response.status(400).entity("Unknown
messageType").build();
    }

    return Response.ok().build();
}

```

```

@POST
@Path("/subscribe")
public Response subscribe() {
    String notificationEndpoint = notificationEndpoint();
    SubscribeResponse response = sns.subscribe(s -> s.topicArn
(topicArn).protocol("http").endpoint(notificationEndpoint));
    subscriptionArn = response.subscriptionArn();
    LOGGER.infov("Subscribed Quarks shield <{0}> : {1} ",
notificationEndpoint, response.subscriptionArn());
    return Response.ok().entity(response.subscriptionArn())
.build();
}

@POST
@Path("/unsubscribe")
public Response unsubscribe() {
    if (subscriptionArn != null) {
        sns.unsubscribe(s -> s.subscriptionArn(subscriptionArn
));
        LOGGER.infov("Unsubscribed quarks shield for id = {0}",
subscriptionArn);
        return Response.ok().build();
    } else {
        LOGGER.info("Not subscribed yet");
        return Response.status(400).entity("Not subscribed yet
").build();
    }
}

private String notificationEndpoint() {
    return quarksShieldBaseUrl + "/sync/shield";
}

private <T> T readObject(Class<T> clazz, String message) {
    T object = null;
    try {
        object = READERS.get(clazz).readValue(message);
    } catch (JsonProcessingException e) {
        LOGGER.errorv("Unable to deserialize message <{0}> to
Class <{1}>", message, clazz.getSimpleName());
        throw new RuntimeException(e);
    }
    return object;
}
}

```

1. `subscribe()` endpoint subscribes to our topic by providing the URL to the POST endpoint receiving SNS notification requests.

2. `unsubscribe()` simply removes our subscription, so any messages published to the topic will not be routed to our endpoint anymore
3. `notificationEndpoint()` is called by SNS on new message if endpoint is subscribed. See [Amazon SNS message and JSON formats](#) for details about the format of the messages SNS can submit.

## Configuring SNS clients

Both SNS clients (sync and async) are configurable via the `application.properties` file that can be provided in the `src/main/resources` directory. Additionally, you need to add to the classpath a proper implementation of the sync client. By default the extension uses URL connection HTTP client, so you need to add a URL connection client dependency to the `pom.xml` file:

```
<dependency>
  <groupId>software.amazon.awssdk</groupId>
  <artifactId>url-connection-client</artifactId>
</dependency>
```

If you want to use Apache HTTP client instead, configure it as follows:

```
quarkus.sns.sync-client.type=apache
```

And add the following dependency to the application `pom.xml`:

```
<dependency>
  <groupId>software.amazon.awssdk</groupId>
  <artifactId>apache-client</artifactId>
</dependency>
```

If you're going to use a local SNS instance, configure it as follows:

```
quarkus.sns.endpoint-override=http://localhost:8009

quarkus.sns.aws.region=us-east-1
quarkus.sns.aws.credentials.type=static
quarkus.sns.aws.credentials.static-provider.access-key-id=test-key
quarkus.sns.aws.credentials.static-provider.secret-access-key=test-secret
```

- `quarkus.sns.aws.region` - It's required by the client, but since you're using a local SNS instance you can pick any valid AWS region.
- `quarkus.sns.aws.credentials.type` - Set `static` credentials provider with any values for

`access-key-id` and `secret-access-key`

- `quarkus.sns.endpoint-override` - Override the SNS client to use a local instance instead of an AWS service

If you want to work with an AWS account, you'd need to set it with:

```
quarkus.sns.aws.region=<YOUR_REGION>
quarkus.sns.aws.credentials.type=default
```

- `quarkus.sns.aws.region` you should set it to the region where you provisioned the SNS table,
- `quarkus.sns.aws.credentials.type` - use the `default` credentials provider chain that looks for credentials in this order:
  - Java System Properties - `aws.accessKeyId` and `aws.secretAccessKey`
  - Environment Variables - `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY`
  - Credential profiles file at the default location (`~/.aws/credentials`) shared by all AWS SDKs and the AWS CLI
  - Credentials delivered through the Amazon ECS if the `AWS_CONTAINER_CREDENTIALS_RELATIVE_URI` environment variable is set and the security manager has permission to access the variable,
  - Instance profile credentials delivered through the Amazon EC2 metadata service

## Next steps

### Packaging

Packaging your application is as simple as `./mvnw clean package`. It can be run with `java -Dtopic.arn=$TOPIC_ARN -jar target/amazon-sns-quickstart-1.0-SNAPSHOT-runner.jar`.

With GraalVM installed, you can also create a native executable binary: `./mvnw clean package -Dnative`. Depending on your system, that will take some time.

### Going asynchronous

Thanks to the AWS SDK v2.x used by the Quarkus extension, you can use the asynchronous programming model out of the box.

Create a `org.acme.sns.QuarksCannonAsyncResource` REST resource that will be similar to our `QuarksCannonSyncResource` but using an asynchronous programming model.

```
package org.acme.sns;

import com.fasterxml.jackson.databind.ObjectMapper;
```

```

import com.fasterxml.jackson.databind.ObjectWriter;
import io.smallrye.mutiny.Uni;
import javax.inject.Inject;
import javax.ws.rs.Consumes;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
import org.acme.sns.model.Quark;
import org.eclipse.microprofile.config.inject.ConfigProperty;
import org.jboss.logging.Logger;
import software.amazon.awssdk.services.sns.SnsAsyncClient;
import software.amazon.awssdk.services.sns.model.PublishResponse;

@Path("/async/cannon")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public class QuarksCannonAsyncResource {

    private static final Logger LOGGER = Logger.getLogger(
        (QuarksCannonAsyncResource.class));

    @Inject
    SnsAsyncClient sns;

    @ConfigProperty(name = "topic.arn")
    String topicArn;

    static ObjectWriter QUARK_WRITER = new ObjectMapper().
        writerFor(Quark.class);

    @POST
    @Path("/shoot")
    @Consumes(MediaType.APPLICATION_JSON)
    public Uni<Response> publish(Quark quark) throws Exception {
        String message = QUARK_WRITER.writeValueAsString(quark);
        return Uni.createFrom()
            .completionStage(sns.publish(p -> p.topicArn(topicArn)
                .message(message)))
            .onItem().invoke(item -> LOGGER.infov("Fired Quark[{0}, {1}]",
                quark.getFlavor(), quark.getSpin()))
            .onItem().apply(PublishResponse::messageId)
            .onItem().apply(id -> Response.ok().entity(id).build())
    };
}

```

We create **Uni** instances from the **CompletionStage** objects returned by the asynchronous SNS



client, and then transform the emitted item.

And corresponding async subscriber `org.acme.sns.QuarksShieldAsyncResource`

```
package org.acme.sns;

import com.fasterxml.jackson.core.JsonProcessingException;
import com.fasterxml.jackson.databind.ObjectMapper;
import com.fasterxml.jackson.databind.ObjectReader;
import io.smallrye.mutiny.Uni;
import java.util.HashMap;
import java.util.Map;
import javax.inject.Inject;
import javax.ws.rs.Consumes;
import javax.ws.rs.HeaderParam;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
import org.acme.sns.model.Quark;
import org.acme.sns.model.SnsNotification;
import org.acme.sns.model.SnsSubscriptionConfirmation;
import org.eclipse.microprofile.config.inject.ConfigProperty;
import org.jboss.logging.Logger;
import software.amazon.awssdk.services.sns.SnsAsyncClient;
import software.amazon.awssdk.services.sns.model.SubscribeResponse;

@Path("/async/shield")
public class QuarksShieldAsyncResource {

    private static final Logger LOGGER = Logger.getLogger(
        QuarksShieldAsyncResource.class);

    private static final String NOTIFICATION_TYPE = "Notification";
    private static final String SUBSCRIPTION_CONFIRMATION_TYPE =
        "SubscriptionConfirmation";
    private static final String UNSUBSCRIPTION_CONFIRMATION_TYPE =
        "UnsubscribeConfirmation";

    @Inject
    SnsAsyncClient sns;

    @ConfigProperty(name = "topic.arn")
    String topicArn;

    @ConfigProperty(name = "quarks.shield.base.url")
    String quarksShieldBaseUrl;

    private volatile String subscriptionArn;
```

```

static Map<Class<?>, ObjectReader> READERS = new HashMap<>();

static {
    READERS.put(SnsNotification.class, new ObjectMapper()
.readerFor(SnsNotification.class));
    READERS.put(SnsSubscriptionConfirmation.class, new
ObjectMapper().readerFor(SnsSubscriptionConfirmation.class));
    READERS.put(Quark.class, new ObjectMapper().readerFor(
Quark.class));
}

@POST
@Consumes({MediaType.TEXT_PLAIN})
public Uni<Response> notificationEndpoint(@HeaderParam("x-amz-
sns-message-type") String messageType, String message) {
    if (messageType == null) {
        return Uni.createFrom().item(Response.status(400).
build());
    }

    if (messageType.equals(NOTIFICATION_TYPE)) {
        return Uni.createFrom().item(readObject(
SnsNotification.class, message))
            .onItem().apply(notification -> readObject(Quark
.class, notification.getMessage()))
            .onItem().invoke(quark -> LOGGER.infov("Quark[{0},
{1}] collision with the shield.", quark.getFlavor(), quark.getSpin
()))
            .onItem().apply(quark -> Response.ok().build());
    } else if (messageType.equals
(SUBSCRIPTION_CONFIRMATION_TYPE)) {
        return Uni.createFrom().item(readObject
(SnsSubscriptionConfirmation.class, message))
            .onItem().produceCompletionStage(msg -> sns
.confirmSubscription(confirm -> confirm.topicArn(topicArn).token
(msg.getToken())))
            .onItem().invoke(resp -> LOGGER.info("Subscription
confirmed. Ready for quarks collisions."))
            .onItem().apply(resp -> Response.ok().build());
    } else if (messageType.equals
(UNSUBSCRIPTION_CONFIRMATION_TYPE)) {
        LOGGER.info("We are unsubscribed");
        return Uni.createFrom().item(Response.ok().build());
    }

    return Uni.createFrom().item(Response.status(400).entity(
"Unknown messageType").build());
}

```

```

@POST
@Path("/subscribe")
public Uni<Response> subscribe() {
    return Uni.createFrom()
        .completionStage(sns.subscribe(s -> s.topicArn(
topicArn).protocol("http").endpoint(notificationEndpoint()))
        .onItem().apply(SubscribeResponse::subscriptionArn)
        .onItem().invoke(this::setSubscriptionArn)
        .onItem().invoke(arn -> LOGGER.infov("Subscribed Quarks
shield with id = {0} ", arn))
        .onItem().apply(arn -> Response.ok().entity(arn).build
()));
}

@POST
@Path("/unsubscribe")
public Uni<Response> unsubscribe() {
    if (subscriptionArn != null) {
        return Uni.createFrom()
            .completionStage(sns.unsubscribe(s -> s
.subscriptionArn(subscriptionArn))
            .onItem().invoke(arn -> LOGGER.infov("Unsubscribed
quarks shield for id = {0}", subscriptionArn))
            .onItem().invoke(arn -> subscriptionArn = null)
            .onItem().apply(arn -> Response.ok().build());
    } else {
        return Uni.createFrom().item(Response.status(400)
.entity("Not subscribed yet").build());
    }
}

private String notificationEndpoint() {
    return quarksShieldBaseUrl + "/async/shield";
}

private void setSubscriptionArn(String arn) {
    this.subscriptionArn = arn;
}

private <T> T readObject(Class<T> clazz, String message) {
    T object = null;
    try {
        object = READERS.get(clazz).readValue(message);
    } catch (JsonProcessingException e) {
        LOGGER.errorv("Unable to deserialize message <{0}> to
Class <{1}>", message, clazz.getSimpleName());
        throw new RuntimeException(e);
    }
}

```

```

    return object;
  }
}

```


And we need to add Netty HTTP client dependency to the `pom.xml`:




```


<dependency>
  <groupId>software.amazon.awssdk</groupId>
  <artifactId>netty-nio-client</artifactId>
</dependency>

```

## Configuration Reference

 Configuration property fixed at build time - All other configuration properties are overridable at runtime

Configuration property	Type	Default
 <code>quarkus.sns.interceptors</code>  List of execution interceptors that will have access to read and modify the request and response objects as they are processed by the AWS SDK. The list should consists of class names which implements <code>software.amazon.awssdk.core.interceptor.ExecutionInterceptor</code> or interface.	list of class name	
 <code>quarkus.sns.sync-client.type</code>  Type of the sync HTTP client implementation	url, apache	url
AWS SDK client configurations	Type	Default
<code>quarkus.sns.endpoint-override</code>  The endpoint URI with which the SDK should communicate. If not specified, an appropriate endpoint to be used for the given service and region.	URI	
<code>quarkus.sns.api-call-timeout</code>  The amount of time to allow the client to complete the execution of an API call. This timeout covers the entire client execution except for marshalling. This includes request handler execution, all HTTP requests including retries, unmarshalling, etc. This value should always be positive, if present.	Duration 	

<code>quarkus.sns.api-call-attempt-timeout</code>  The amount of time to wait for the HTTP request to complete before giving up and timing out. This value should always be positive, if present.	Duration 	
<b>AWS services configurations</b>	<b>Type</b>	<b>Default</b>
<code>quarkus.sns.aws.region</code>  An Amazon Web Services region that hosts the given service.  It overrides region provider chain with static value of region with which the service client should communicate.  If not set, region is retrieved via the default providers chain in the following order: <ul style="list-style-type: none"> <li>• <code>aws.region</code> system property</li> <li>• <code>region</code> property from the profile file</li> <li>• Instance profile file</li> </ul> See <code>software.amazon.awssdk.regions.Region</code> for available regions.	Region	

`quarkus.sns.aws.credentials.type`


Configure the credentials provider that should be used to authenticate with AWS.

Available values:

- **default** - the provider will attempt to identify the credentials automatically using the following checks:
  - Java System Properties - `aws.accessKeyId` and `aws.secretKey`
  - Environment Variables - `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY`
  - Credential profiles file at the default location (`~/.aws/credentials`) shared by all AWS SDKs and the AWS CLI
  - Credentials delivered through the Amazon EC2 container service if `AWS_CONTAINER_CREDENTIALS_RELATIVE_URI` environment variable is set and security manager has permission to access the variable.
  - Instance profile credentials delivered through the Amazon EC2 metadata service
- **static** - the provider that uses the access key and secret access key specified in the `static-provider` section of the config.
- **system-property** - it loads credentials from the `aws.accessKeyId`, `aws.secretAccessKey` and `aws.sessionToken` system properties.
- **env-variable** - it loads credentials from the `AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY` and `AWS_SESSION_TOKEN` environment variables.
- **profile** - credentials are based on AWS configuration profiles. This loads credentials from a [profile file](#), allowing you to share multiple sets of AWS security credentials between different tools like the AWS SDK for Java and the AWS CLI.
- **container** - It loads credentials from a local metadata service. Containers currently supported by the AWS SDK are **Amazon Elastic Container Service (ECS)** and **AWS Greengrass**
- **instance-profile** - It loads credentials from the Amazon EC2 Instance Metadata Service.
- **process** - Credentials are loaded from an external process. This is used to support the `credential_process` setting in the profile credentials file. See [Sourcing Credentials From External Processes](#) for more information.
- **anonymous** - It always returns anonymous AWS credentials. Anonymous AWS credentials result in un-authenticated requests and will fail unless the resource or API's policy has been configured to specifically allow anonymous access.

default,  
static,  
system-  
prop-  
erty,  
env-  
variab  
le,  
profil  
e,  
contai  
ner,  
instan  
ce-  
profil  
e,  
proces  
s,  
anonym  
ous

default

Default credentials provider configuration	Type	Default
<code>quarkus.sns.aws.credentials.default-provider.async-credential-update-enabled</code>  Whether this provider should fetch credentials asynchronously in the background. If this is <code>true</code> , threads are less likely to block, but additional resources are used to maintain the provider.	boolean	<code>false</code>
<code>quarkus.sns.aws.credentials.default-provider.reuse-last-provider-enabled</code>  Whether the provider should reuse the last successful credentials provider in the chain. Reusing the last successful credentials provider will typically return credentials faster than searching through the chain.	boolean	<code>true</code>
Static credentials provider configuration	Type	Default
<code>quarkus.sns.aws.credentials.static-provider.access-key-id</code>  AWS Access key id	string	
<code>quarkus.sns.aws.credentials.static-provider.secret-access-key</code>  AWS Secret access key	string	
AWS Profile credentials provider configuration	Type	Default
<code>quarkus.sns.aws.credentials.profile-provider.profile-name</code>  The name of the profile that should be used by this credentials provider. If not specified, the value in <code>AWS_PROFILE</code> environment variable or <code>aws.profile</code> system property is used and defaults to <code>default</code> name.	string	
Process credentials provider configuration	Type	Default
<code>quarkus.sns.aws.credentials.process-provider.async-credential-update-enabled</code>  Whether the provider should fetch credentials asynchronously in the background. If this is true, threads are less likely to block when credentials are loaded, but additional resources are used to maintain the provider.	boolean	<code>false</code>
<code>quarkus.sns.aws.credentials.process-provider.credential-refresh-threshold</code>  The amount of time between when the credentials expire and when the credentials should start to be refreshed. This allows the credentials to be refreshed <b>before</b> they are reported to expire.	Duration 	<code>15S</code>

<code>quarkus.sns.aws.credentials.process-provider.process-output-limit</code> The maximum size of the output that can be returned by the external process before an exception is raised.	Memory Size ?	1024
<code>quarkus.sns.aws.credentials.process-provider.command</code> The command that should be executed to retrieve credentials.	string	
<b>Sync HTTP transport configurations</b>	<b>Type</b>	<b>Default</b>
<code>quarkus.sns.sync-client.connection-timeout</code> The maximum amount of time to establish a connection before timing out.	Duration ?	2S
<code>quarkus.sns.sync-client.socket-timeout</code> The amount of time to wait for data to be transferred over an established, open connection before the connection is timed out.	Duration ?	30S
<b>Apache HTTP client specific configurations</b>	<b>Type</b>	<b>Default</b>
<code>quarkus.sns.sync-client.apache.connection-acquisition-timeout</code> The amount of time to wait when acquiring a connection from the pool before giving up and timing out.	Duration ?	10S
<code>quarkus.sns.sync-client.apache.connection-max-idle-time</code> The maximum amount of time that a connection should be allowed to remain open while idle.	Duration ?	60S
<code>quarkus.sns.sync-client.apache.connection-time-to-live</code> The maximum amount of time that a connection should be allowed to remain open, regardless of usage frequency.	Duration ?	
<code>quarkus.sns.sync-client.apache.max-connections</code> The maximum number of connections allowed in the connection pool. Each built HTTP client has its own private connection pool.	int	50
<code>quarkus.sns.sync-client.apache.expect-continue-enabled</code> Whether the client should send an HTTP expect-continue handshake before each request.	boolean	true



<code>quarkus.sns.sync-client.apache.use-idle-connection-reaper</code>  Whether the idle connections in the connection pool should be closed asynchronously. When enabled, connections left idling for longer than <code>quarkus..sync-client.connection-max-idle-time</code> will be closed. This will not close connections currently in use.	boolean	<code>true</code>
<code>quarkus.sns.sync-client.apache.proxy.enabled</code>  Enable HTTP proxy	boolean	<code>false</code>
<code>quarkus.sns.sync-client.apache.proxy.endpoint</code>  The endpoint of the proxy server that the SDK should connect through. Currently, the endpoint is limited to a host and port. Any other URI components will result in an exception being raised.	URI	
<code>quarkus.sns.sync-client.apache.proxy.username</code>  The username to use when connecting through a proxy.	string	
<code>quarkus.sns.sync-client.apache.proxy.password</code>  The password to use when connecting through a proxy.	string	
<code>quarkus.sns.sync-client.apache.proxy.ntlm-domain</code>  For NTLM proxies - the Windows domain name to use when authenticating with the proxy.	string	
<code>quarkus.sns.sync-client.apache.proxy.ntlm-workstation</code>  For NTLM proxies - the Windows workstation name to use when authenticating with the proxy.	string	
<code>quarkus.sns.sync-client.apache.proxy.preemptive-basic-authentication-enabled</code>  Whether to attempt to authenticate preemptively against the proxy server using basic authentication.	boolean	
<code>quarkus.sns.sync-client.apache.proxy.non-proxy-hosts</code>  The hosts that the client is allowed to access without going through the proxy.	list of string	

<code>quarkus.sns.sync-client.apache.tls-managers-provider.type</code> TLS managers provider type. Available providers: <ul style="list-style-type: none"> <li>• <b>none</b> - Use this provider if you don't want the client to present any certificates to the remote TLS host.</li> <li>• <b>system-property</b> - Provider checks the standard <code>javax.net.ssl.keyStore</code>, <code>javax.net.ssl.keyStorePassword</code>, and <code>javax.net.ssl.keyStoreType</code> properties defined by the <a href="#">JSSE</a>.</li> <li>• <b>file-store</b> - Provider that loads a the key store from a file.</li> </ul>	none, system- -prop erty, file- store	system- -prop erty
<code>quarkus.sns.sync-client.apache.tls-managers-provider.file-store.path</code> Path to the key store.	path	
<code>quarkus.sns.sync-client.apache.tls-managers-provider.file-store.type</code> Key store type. See the KeyStore section in the <a href="#">Java Cryptography Architecture Standard Algorithm Name Documentation</a> for information about standard keystore types.	string	
<code>quarkus.sns.sync-client.apache.tls-managers-provider.file-store.password</code> Key store password	string	
<b>Netty HTTP transport configurations</b>	<b>Type</b>	<b>Default</b>
<code>quarkus.sns.async-client.max-concurrency</code> The maximum number of allowed concurrent requests. For HTTP/1.1 this is the same as max connections. For HTTP/2 the number of connections that will be used depends on the max streams allowed per connection.	int	<b>50</b>
<code>quarkus.sns.async-client.max-pending-connection-acquires</code> The maximum number of pending acquires allowed. Once this exceeds, acquire tries will be failed.	int	<b>10000</b>
<code>quarkus.sns.async-client.read-timeout</code> The amount of time to wait for a read on a socket before an exception is thrown. Specify <b>0</b> to disable.	Duration 	<b>30S</b>

<code>quarkus.sns.async-client.write-timeout</code>	Duration ?	30S
The amount of time to wait for a write on a socket before an exception is thrown. Specify 0 to disable.		
<code>quarkus.sns.async-client.connection-timeout</code>	Duration ?	10S
The amount of time to wait when initially establishing a connection before giving up and timing out.		
<code>quarkus.sns.async-client.connection-acquisition-timeout</code>	Duration ?	2S
The amount of time to wait when acquiring a connection from the pool before giving up and timing out.		
<code>quarkus.sns.async-client.connection-time-to-live</code>	Duration ?	
The maximum amount of time that a connection should be allowed to remain open, regardless of usage frequency.		
<code>quarkus.sns.async-client.connection-max-idle-time</code>	Duration ?	60S
The maximum amount of time that a connection should be allowed to remain open while idle. Currently has no effect if <code>quarkus..async-client.use-idle-connection-reaper</code> is false.		
<code>quarkus.sns.async-client.use-idle-connection-reaper</code>	boolean	true
Whether the idle connections in the connection pool should be closed. When enabled, connections left idling for longer than <code>quarkus..async-client.connection-max-idle-time</code> will be closed. This will not close connections currently in use.		
<code>quarkus.sns.async-client.protocol</code>	http1-1, http2	http1-1
The HTTP protocol to use.		
<code>quarkus.sns.async-client.ssl-provider</code>	jdk, openssl, openssl- refcnt	
The SSL Provider to be used in the Netty client. Default is <code>OPENSSL</code> if available, <code>JDK</code> otherwise.		
<code>quarkus.sns.async-client.http2.max-streams</code>	long	4294967295
The maximum number of concurrent streams for an HTTP/2 connection. This setting is only respected when the HTTP/2 protocol is used.		

<code>quarkus.sns.async-client.http2.initial-window-size</code>  The initial window size for an HTTP/2 stream. This setting is only respected when the HTTP/2 protocol is used.	int	1048576
<code>quarkus.sns.async-client.proxy.enabled</code>  Enable HTTP proxy.	boolean	false
<code>quarkus.sns.async-client.proxy.endpoint</code>  The endpoint of the proxy server that the SDK should connect through. Currently, the endpoint is limited to a host and port. Any other URI components will result in an exception being raised.	URI	
<code>quarkus.sns.async-client.proxy.non-proxy-hosts</code>  The hosts that the client is allowed to access without going through the proxy.	list of string	
<code>quarkus.sns.async-client.tls-managers-provider.type</code>  TLS managers provider type.  Available providers: <ul style="list-style-type: none"> <li>• <b>none</b> - Use this provider if you don't want the client to present any certificates to the remote TLS host.</li> <li>• <b>system-property</b> - Provider checks the standard <code>javax.net.ssl.keyStore</code>, <code>javax.net.ssl.keyStorePassword</code>, and <code>javax.net.ssl.keyStoreType</code> properties defined by the <a href="#">JSSE</a>.</li> <li>• <b>file-store</b> - Provider that loads a the key store from a file.</li> </ul>	none, system-property, file-store	system-property
<code>quarkus.sns.async-client.tls-managers-provider.file-store.path</code>  Path to the key store.	path	
<code>quarkus.sns.async-client.tls-managers-provider.file-store.type</code>  Key store type. See the KeyStore section in the <a href="#">Java Cryptography Architecture Standard Algorithm Name Documentation</a> for information about standard keystore types.	string	

<code>quarkus.sns.async-client.tls-managers-provider.file-store.password</code> Key store password	string	
<code>quarkus.sns.async-client.event-loop.override</code> Enable the custom configuration of the Netty event loop group.	boolean	<code>false</code>
<code>quarkus.sns.async-client.event-loop.number-of-threads</code> Number of threads to use for the event loop group. If not set, the default Netty thread count is used (which is double the number of available processors unless the <code>io.netty.eventLoopThreads</code> system property is set.	int	
<code>quarkus.sns.async-client.event-loop.thread-name-prefix</code> The thread name prefix for threads created by this thread factory used by event loop group. The prefix will be appended with a number unique to the thread factory and a number unique to the thread. If not specified it defaults to <code>aws-java-sdk-NettyEventLoop</code>	string	



#### *About the Duration format*

The format for durations uses the standard `java.time.Duration` format. You can learn more about it in the [Duration#parse\(\) javadoc](#).

You can also provide duration values starting with a number. In this case, if the value consists only of a number, the converter treats the value as seconds. Otherwise, `PT` is implicitly prepended to the value to obtain a standard `java.time.Duration` format.



#### *About the MemorySize format*

A size configuration option recognises string in this format (shown as a regular expression): `[0-9]+[KkMmGgTtPpEeZzYy]?`. If no suffix is given, assume bytes.